NASA-TM-111475

7 6

Localized Search for Controlling Automated Reasoning

AMY L. LANSKY

AI RESEARCH BRANCH, MAIL STOP 244-17
NASA AMES RESEARCH CENTER
MOFFETT FIELD, CA 94035

NASA Ames Research Center

Artificial Intelligence Research Branch

Technical Report FIA-90-08-15-01

August, 1990

Localized Search
for
Controlling Automated Reasoning

Amy L. Lansky

Sterling Software/NASA Ames Research Center
Artificial Intelligence Research Branch
MS 244-17, Moffett Field, CA 94035 USA
LANSKY@PLUTO.ARC.NASA.GOV

August 15, 1990

Abstract

This paper describes the localized search mechanism of the GEMPLAN multiagent planner. Both formal complexity results and empirical results are provided, demonstrating the benefits of localized search. Localized search utilizes an explicit domain decomposition to infer constraint localization semantics and, as a result, to enable the decomposition of the planning search space into a set of spaces, one for each domain region. Each search tree is concerned with the construction of a region plan that satisfies regional constraints. Shifts between search trees are guided by potential regional interactions as defined by the domain's structure. The search algorithm must also ensure that all search trees are consistent, which is especially difficult in the case of regional overlap. Benefits of localization include a smaller and cheaper overall search space as well as heuristic guidance in controlling search. Such benefits are critical if current planning technologies are to be scaled up to large, complex domains. Indeed, the use of domain localization and localized search are broadly applicable techniques that can be used by many kinds of domain reasoning systems, not just planners.

A slightly shortened version of this paper was submitted to the 1990 DARPA Planning Workshop, San Diego, California, November 1990. This research has been made possible in part by the National Science Foundation, under Grant IRI-8715972.

Contents

1 Introduction

By now, the algorithmic complexity of domain-independent planning has become well known [2]. Many planning researchers have given up completely on pre-planning frameworks, opting, instead, for more reactive action-generation strategies. Yet, there are many domains for which purely reactive approaches are inadequate. Imagine, for example, a factory shop floor in which people coordinate their activities simply by dynamically "reacting" to one another. The shop floor would soon become a mess. Some amount of pre-planning is necessary for domains that require complex coordination of activities, especially when adherence to coordination constraints is critical. Such domains are numerous and include NASA mission planning[1], building-construction planning, factory planning, and other forms of organizational planning and coordination. Given the inescapable need for reasoning about large complex plans, the planning community faces two related obstacles: (1) the inherent costliness of plan construction algorithms and (2) the problem of scaling planning systems up to large domains. Indeed, these obstacles pose problems for *any* form of planning, whether it is performed before or during execution.

The focus of this paper is the use of *locality* — the inherent structural properties of a domain — to control the explosive cost of planning and other forms of reasoning. The use of localized reasoning, while quite intuitive and natural, has not been a fundamental aspect of most AI systems. A localized domain description is one that is explicitly decomposed into a set of regions. Each region may be viewed as a subset of domain activity with an associated set of "constraints" (properties, goals, or other requirements that the planner must fulfill) that are applicable only to the activities within the region. We refer to this delineation of constraint applicability as *constraint localization. Localized planning* consists of searching a set of smaller, regional planning search spaces rather than a large, "global" space. Each GEMPLAN search space may be visualized as a regional plan-construction search tree, where each tree node is associated with a region plan and each arc is associated with a plan modification that transforms the preceding plan into a new plan (with new actions, relationships, etc.) that satisfies the constraint

[illegible]

The GEMPLAN domain representation allows for the specification of any kind of domain decomposition, including the use of regions that overlap, are disjoint, are organized hierarchically, or form any combination thereof. Criteria for decomposition are usually suggested by the innate characteristics of a domain, such as its physical structure, its behavioral entities (agents), and its functional elements. Indeed, it is often useful to utilize a decomposition that reflects several criteria simultaneously. Consider, for example, a building-construction domain. Viewed globally, the domain may be described by a set of constraints, some of which describe the actual structure and requirements for a specific building, some that encode the requirements and capabilities of contractors and physi-

[1]Throughout the rest of this paper, the term "planning" will be used rather than "planning and schedul

already utilize heuristics that are provided by domain structure. The localized search algorithm described in this paper could thus be applied to many kinds of reasoning systems. However, we do stress multiagent domains here for two reasons: (1) the complexity of coordinating multiagent domains makes localization even more necessary; (2) multiagent domains are typically easy to decompose.

Of course, the benefits of localized search do have a price. Most realistic domains cannot be partitioned into simple hierarchies or disjoint regions. Domains of any complexity will have regions that "overlap" – that is, some subregions will be shared by more than one encompassing region. For example, in Figure 1, regions wallA, e-control, and p-control each belong to more than one region. This complicates the localized search algorithm because changes within a shared region must be reflected within the search trees of all its ancestors. That is, localized search must pay attention to the problem of *interaction and consistency* among search trees.

In addition, constraint localization will yield large gains only if a domain can be effectively decomposed. If many constraints naturally belong to a region that includes a great deal of domain activity, search will remain quite expensive. To gain real efficiency benefits, regions which may seem intuitively "global" should be composed to include only a minimal amount of activity. For example, in Figure 1, the general contractor's constraints apply only to his/her own activities in gc-control, those in e-control (electrician control activities), and those in p-control (plumber control activities), not to *all* activities in electrician and plumber. Experience thus far with GEMPLAN (and commonsense intuition about the

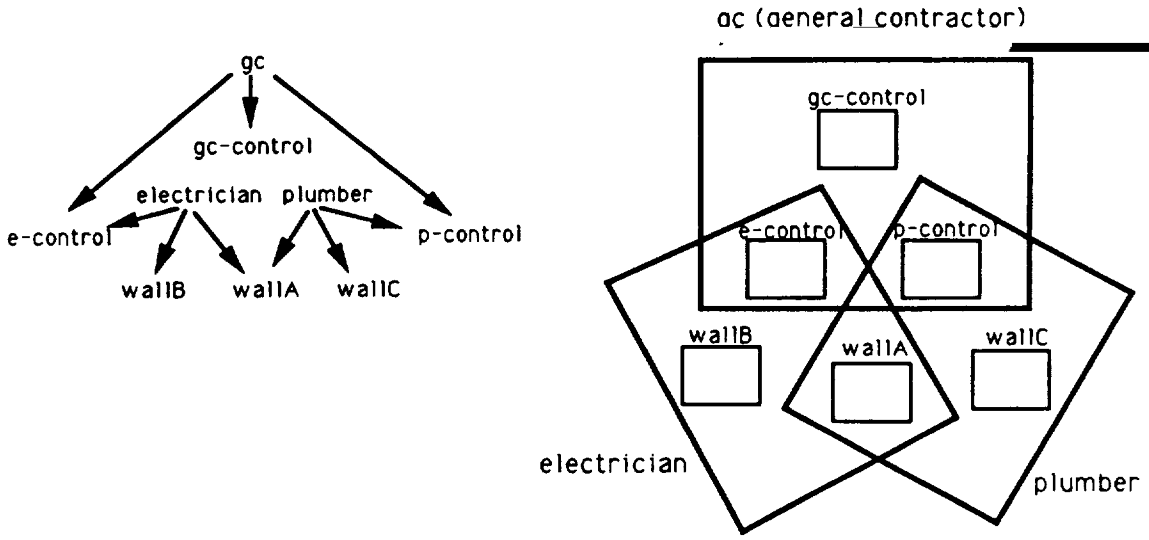


Figure 1: A Localized Construction Domain Description

- ▷ GEMPLAN partitions the global search space into localized search spaces.

- ▷ GEMPLAN has a highly flexible, tailorable search mechanism. In particular, constraint application can be guided by user-supplied heuristics and by the changing planning and/or execution context, as it develops.

- ▷ GEMPLAN has the ability to satisfy a broad range of domain "constraint forms," not simply the attainment and maintenance of state conditions (the traditional notion of "planning"). Note that we are using the term "constraint" in a very broad sense – i.e., to label any sort of property that the planner knows how to test and to make true. The system includes a set of general-purpose constraint satisfaction algorithms for partially-ordered plans, which may be further augmented by user-supplied constraint-satisfaction methods. The default constraint algorithms are fully general – they allow for the addition of and reasoning about any possible temporal relationship between actions, including simultaneity. It is these constraint satisfaction algorithms that perform the task of plan construction and coordination, by introducing actions, action interrelationships, and variable bindings. The current constraint repertoire includes:

  - The attainment and maintenance of goal conditions and preconditions – i.e., the traditional "planning algorithm." Actions may be defined to have conditional effects. The algorithm also includes full protection capabilities.

- Action decomposition (i.e., action hierarchies). GEMPLAN allows for reasoning about actions at mixed levels of detail, rather than confining itself to reasoning "one level at a time," as do some hierarchical planners [13]. Indeed, rather than being inextricably bound to the planner's search structure (hiearchical or otherwise), action decomposition is just another kind of "constraint" to be satisfied by the system, and may be applied at any appropriate time, including at run-time. For instance, run-time action decomposition would result in planning behavior much that like displayed by reactive systems such as PRS [3].
- A variety of temporal and causal constraints, including run-time priority constraints such as "first-come-first-serve."
- Attainment of desired patterns of behavior expressed as regular expressions.

More details on GEMPLAN appear elsewhere [7,8,9,10]. The rest of this paper will focus on GEMPLAN's localized search mechanism. A specific example of localized plan construction is presented in Section 3.

2.1 Search Space Decomposition

As described earlier, a GEMPLAN domain specification is decomposed with the goal of localizing the applicability of constraints as much as possible. For example, the construction domain depicted in Figure 1 has been partitioned into regions corresponding to the activities of the electrician, plumber, and general contractor. These regions have been further decomposed to include subregions that contain the activities of the electrician and plumber at various walls as well as contractor "control" activities (these might include communication actions or high-level actions that have not yet been expanded into activities at particular walls). Each wall region would be associated with constraints and definitions that are relevant to the actions taking place at that wall. For example, in the case of wallA, these may include constraints relating to coordination of plumber and electrician activities. Each control region might be associated with personal communication and planning constraints for that contractor. The electrician, plumber, and gc region constraints, which apply to all their subregions, might describe more global requirements pertinent to their respective activities. For example, note how the gc constraints apply to all the control regions. These might describe how the general contractor's requests influence the control activities of each subcontractor.

Rather than searching a single global search space, GEMPLAN creates a regional search space for each region. Each search space is concerned with building a plan for its region that satisfies all regional constraints. The planner may thus be viewed as a set of "mini-planners," tied together as dictated by the structural relationships between regions.

electrician
wallB
Plan-i
check constraint

many possible solutions or "fixes" per fix algorithm), resulting in a set of new region plans at the next level down in the tree. A GEMPLAN fix typically adds new actions, relations, and variable bindings to a region plan, and may also generate new subregions. Note that fixes may add actions and relations *anywhere* within the plan it is working on – the precise temporal position is determined by the nature of the constraint and fix.[3]

GEMPLAN uses, by default, a depth-first search strategy for searching its trees, trying constraints in the order supplied by the domain specification and fixes in the order supplied by GEMPLAN's internal constraint mechanism. However, since search should optimally be driven by domain-dependent information and the structure of the plan itself, GEMPLAN allows for flexible user-tuning of tree search. The order in which constraints and fixes are applied can be made context dependent. GEMPLAN also includes a facility that can determine precisely which actions affect which constraints *within* a region. This facility enables *only* relevant constraints to be applied at each step, thereby exceeding the kind of "frame" information already provided by constraint-localization semantics. This coupling of localized search, where only relevant constraints are checked, with further user-tailoring of the search, forms an extremely flexible mechanism of "relevancy-driven-search" – namely, search driven by the most relevant constraints at any particular time in the reasoning process.

2.4 Plan Representation

As stated above, each search tree node is associated with a *region plan.* Each region plan consists of a *local region plan* and a set of *subplans* (the region plans of its subregions). For example, if $R1 \subset R$ and $R2 \subset R$, the region plan for R will include a local region plan for R and region plans for $R1$ and $R2$. GEMPLAN associates all plan information with the smallest region that encompasses that information. The region plans of $R1$ and $R2$ will thus include all actions, temporal and causal relations, variable bindings, and other plan information that deal exclusively with $R1$ and $R2$, respectively. The local region plan of R will then include plan information that deals specifically with activities in R or that pertains to relationships among R, $R1$, and $R2$ (and therefore cannot be associated strictly with $R1$ or $R2$).

2.5 Guiding Search Among Regional Trees

Search within $tree(R)$ is concerned with (1) assuring that all of R's constraints are satisfied by R's region plan and (2) making sure that R's subregions' trees are searched to find

[3]For example, unlike some planners (typically, those those perform state-space search), actions need not be added to the plan in an order that is in any way related to the order in which the actions are executed. The fix algorithms may thus be viewed generically as plan modifiers that grow and refine plans in flexible ways.

a satisfactory plan for their region plans. Referring back to Figure 1, it is the role of *tree*(electrician) to make sure that electrician's constraints are satisfied and that *tree*(wallB), *tree*(wallA), and *tree*(e-control) are all visited when their subplans may be affected and their constraints need to be rechecked.

How does control transfer among regional trees? This is done in response to information transmitted to the search mechanism by a fix. Suppose we are in *tree*(R). After applying a fix for one of R's constraints to R's region plan, the fix must return a subset of R's subregions, $R1, \ldots, Rm$, that may have been affected by the fix. The GEMPLAN search algorithm will then inhibit further search within *tree*(R) until *tree*($R1$)...*tree*(Rm) are all satisfactorily searched. As depicted in Figure 2, if electrician affects the subplan for region wallB via the introduction of new actions there, search within *tree*(electrician) cannot safely proceed until wallB's tree is searched and its constraints are rechecked and satisfied. Notice how shifts between parent and child regions induce a partitioning on the child's search tree. We call these search fragments *incarnations* – search within the child is "reincarnated" each time its constraints are potentially violated due to a fix in its parent's search tree. Each incarnation is thus a subtree initiated by a parent region. In our example, *tree*(wallB) may be reincarnated several times due to fixes for electrician constraints. Each time *tree*(wallB) is revisited, wallB's constraints must be rechecked and satisfied. One restriction on GEMPLAN's search control mechanism is that all search strategies (e.g., breadth-first, dependency-directed, etc.) must be applied within the confines of an individual incarnation. This greatly simplifies the problem of search consistency.

As the reader may have noticed, not all regions are subregions of some enclosing region. In the domain of Figure 1, this is true of gc, electrician, and plumber. To simplify search, GEMPLAN requires that all tree search ultimately flows from some designated "global" regional tree.[4] Although gc, electrician, and plumber do not logically belong to another region as far as constraint applicability, we do need to make sure that some region at least takes "responsibility" for invoking their search trees. Thus, we include the additional relation $\subset_r$ to denote this relationship, and require that each region except some designated "global" region have a "parent" that assumes search responsibility for it. In our example, we shall choose gc as the "global" region, with electrician $\subset_r$ gc and plumber $\subset_r$ gc. Although *tree*(gc) must make sure that *tree*(electrician) and *tree*(plumber) are visited appropriately, gc's constraints apply only to its region plan, which includes only the subregion plans of gc-control, e-control, and p-control.[5]

[4]This does not preclude the possibility of parallel search of independent subtrees. Our research plans include experimentation with parallel search in GEMPLAN.

[5]Readers of previous papers on GEMPLAN will recall that the GEMPLAN description language includes several types of regions and modes of access between regions (elements, groups, ports, etc.). For the purposes of this paper and for the sake of generality, we simplified the GEMPLAN structural model to include only the relations $\subset$ and $\subset_r$. The semantics of elements, groups, and ports can all be captured in terms of $\subset$ and $\subset_r$.

2.6 Dealing With Regional Overlap

One of the challenges of localized search is keeping all regional search trees consistent with each other. This would be fairly straightforward if domain structure were strictly hierarchical. However, since we allow for regional overlap, some effort is required to keep trees consistent. For example, if a fix in *tree*(electrician) affects region wallA's plan, it is not enough to simply recheck wallA's constraints and return to *tree*(electrician). Region plumber's representation of wallA's subplan must also be updated within *tree*(plumber), and search must also occur within *tree*(plumber) to recheck its constraints. We call this process of maintaining consistency *completion*. Because GEMPLAN allows for quite complex localization structures, the search algorithm must be very careful to perform completion fully and correctly. GEMPLAN must update all affected data structures (in particular, parent tree data) each time it completes searching an incarnation of a shared region. It must also make sure that all affected parent region trees are ultimately reincarnated and that region constraints are rechecked.

3 Example

In this section, we attempt to clarify the preceding discussion with an example from the very simple construction domain of Figure 1. Let us assume that the electrician, plumber, wallA, and wallB regions are associated with the following (informally described) constraints:[6][7]

```
ELECTRICIAN CONSTRAINTS:
(1) action(install-socket(wallA,locA1))
(2) action(install-socket(wallB,locB1))
(3) action(install-socket(wallB,locB2))
(4) decompose(install-socket(W,L), {W.electprep(L) => W.insertsocket(L)})

PLUMBER CONSTRAINTS:
(1) action(install-pipe(wallA,locA1))
(2) decompose(install-pipe(W,L), {W.plumbprep(L) => W.insertpipe(L)})

WALLA CONSTRAINTS:
(1) (forall L)[(forall prep:{electprep(L),plumbprep(L)}) pattern((prep)*=>)]
(2) fcfs([[electprep,insertsocket],[plumbprep,insertpipe]])
```

[6]Tokens starting with a capital letter denote variables.

[7]This toy description is intended for explicatory purposes only. A full GEMPLAN domain description requires specification of potential regional event types and domain structure, allows for specification and instantiation of region types, and may include information about search heuristics as well as other kinds of domain-specific information.

```
WALLB CONSTRAINTS:
(1) all-matching-precede(electprep,insertsocket)
```

The first three electrician constraints require that actions exist in the final plan that install sockets in particular walls and locations. Such `action` constraints simply result in the addition of actions to the plan. The fourth `decompose` constraint requires that each `install-socket(W,L)` action be decomposed into an `electprep` action followed by an `insertsocket` action at wall `W`, location `L`. Note that an action of form `X.Y` denotes an action `Y` occurring at location `X`. The plumber constraints are similar. In this case, only one pipe is to be installed at wallA.[8]

The two wallA constraints pertain to the coordination of the electrician and plumber actions at that wall. The first constraint states that, at wallA, all `electprep` and `plumbprep` actions at the same location follow a certain pattern – they must be totally ordered by the temporal relation `=>`. The second constraint additionally requires that the electrician and plumber have access to wallA on a first-come-first-serve basis. The constraint description consists of a set of constraint pairs and has the following semantics: any required execution ordering of the first actions in each pair (in this case, required orderings between "prep" actions) will determine the ultimate ordering of the second actions in each pair (in this case, the ordering of `insertsocket` and `insertpipe` actions). Since a total ordering is forced on all "prep" actions at the same location, this will force electricians and plumbers who wish to insert their devices in common locations to do so on a first-come-first-serve basis. Finally,

that all electrical wall-prep at wallB will be completed before any electrical components are inserted. At wallA, in contrast, prep and insertion actions may be intermingled, as long as they conform to the two ordering constraints of wallA.

$$\begin{tikzcd}
\texttt{install-socket(wallA,locA1)} \arrow[r, phantom, "\ll"] & \texttt{wallA.electprep(locA1)} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \texttt{wallA.insertsocket(locA1)} \arrow[d, Rightarrow] \\
\texttt{install-pipe(wallA,locA1)} \arrow[r, phantom, "\ll"] & \texttt{wallA.plumbprep(locA1)} \arrow[r, Rightarrow] & \texttt{wallA.insertpipe(locA1)} \\
\texttt{install-socket(wallB,locB1)} \arrow[r, phantom, "\ll"] & \texttt{wallB.electprep(locB1)} \arrow[r, Rightarrow] \arrow[dr, Rightarrow] & \texttt{wallB.insertsocket(locB1)} \\
\texttt{install-socket(wallB,locB2)} \arrow[r, phantom, "\ll"] & \texttt{wallB.electprep(locB2)} \arrow[r, Rightarrow] \arrow[ur, Rightarrow] & \texttt{wallB.insertsocket(locB2)}
\end{tikzcd}$$

Figure 3: A Construction Plan

continues within *tree*(electrician), search within *tree*(wallA) and *tree*(wallB) must occur. Let us assume that wallA is searched first. Both wallA constraints are checked, but both are satisfied at this point. The newly completed incarnation of wallA therefore does not add any new information to the subplan for wallA associated with electrician, but the process of *completion* causes the new version of the wallA plan (that includes the changes made by electrician) to be inserted appropriately into nodes of *tree*(plumber).

Then *tree*(wallB) is searched. The wallB constraint causes the relations `electprep(locB1) => insertsocket(locB2)` and `electprep(locB2) => insertsocket(locB1)` to be added. Search then returns to electrician, and the electrician's subplan for wallB is appropriately updated. Note that wallB is not a region of overlap, so no other completion operation need occur.

At this point, all electrician constraints are satisfied. Search then bounces back to gc, which invokes search in *tree*(plumber). The plumber constraints cause the addition of the `install-pipe` action and its decomposition into the appropriate subactions at wallA. After fixing the second plumber constraint, search must occur for the affected wallA region. This causes the actions `electprep(locA1)` and `plumbprep(locA1)` to be forced into some total order (in Figure 3, `electprep(locA1) => plumbprep(locA1)` was chosen) and then, as a result of the second wallA constraint, a similar ordering is imposed on `insertsocket(locA1)` and `insertpipe(locA1)`. The now satisfied wallA plan is appropriately inserted into both *tree*(electrician) and *tree*(plumber) (due to the completion process). All plumber constraints are now satisfied and search bounces back to gc. The constraints within electrician are then rechecked (due to the changes at wallA), but they are still satisfied. Search then terminates successfully.

Next, search proceeds from that node, repeatedly choosing constraints, applying fixes, and visiting new nodes until a node is reached in which either: all constraints have been satisfied (`satisfied(Node)`), there remain constraints to be satisfied but no possible fix works (`failure(Node)`), or search has for some reason been told to stop (`stop(Node)`). This last option is useful because heuristics might determine that it is best to wait and see if a later incarnation can provide a situation in which remaining constraints can be satisfied more easily. The function `select_next_node` is given a node and selects a new node from which to search. Together with `select_first_node`, it implements the search strategy within an incarnation.

The procedure `apply_constraint_and_fix` utilizes the current state of `Node` to decide whether to:

1. Select a new constraint to check and, if it is not satisfied, apply a fix to obtain a resulting `TempNewNode` (this will occur the first time `Node` is visited);

2. Retry a new fix for a particular constraint or try checking a different constraint (this may occur upon backtracking to `Node`);

3. Backtrack within subregional incarnations (this may occur if search backtracks to `Node` and `Node` induced search at subregional trees);

4. Backtrack within this incarnation (nothing remains to be done at `Node`); or

5. Stop further search in the incarnation, resulting in `stop(Node)`.

As shown, procedure `apply_constraint_and_fix` returns a set `AffectedSubregions`, all of whose subtrees must be visited in order to successfully proceed. The procedure `check_affected_subregions` takes a list of subregions $R_1 \dots R_m$, forms incarnations for them, and searches these incarnations in sequence. If search within R_i fails, R_{i-1}'s incarnation will be retried to find an alternate plan, after which R_i's initial incarnation will be started again. If no combination of incarnations for the subregions can be found, `SubregionResult` will indicate failure. Note that the subregion incarnations will be retried only in the order returned by `apply_constraint_and_fix`. Thus, the search is not complete - i.e., it doesn't invoke the subregional trees in all possible orders. We chose this strategy in order to allow the constraint-fixing algorithms to provide heuristic search guidance by forcing a particular ordering on subregional checking. Note that `TempNewNode` will be updated to become `NewNode`, which reflects changes within the subregions due to search within the subregional trees.

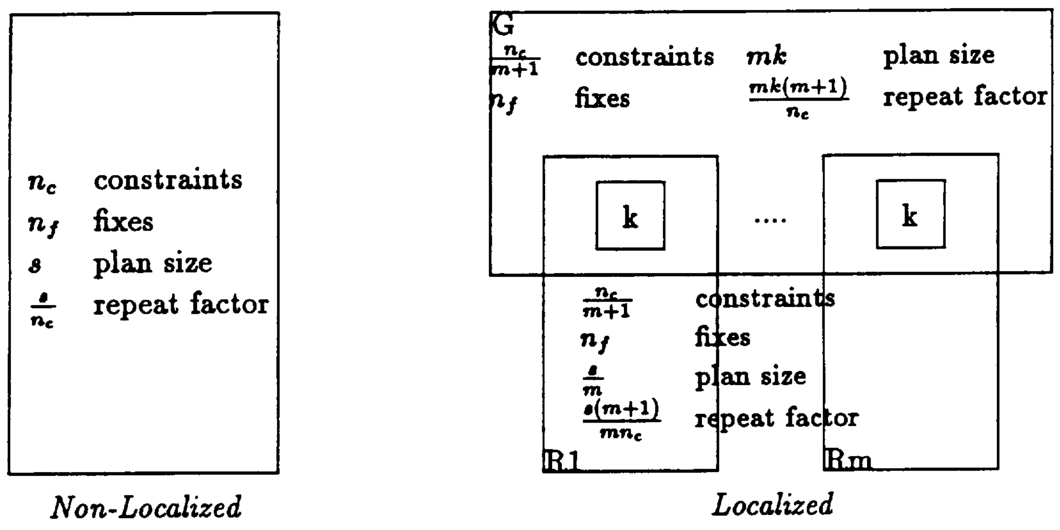
Finally, we have the completion operation `do_completion`. When search within an incarnation for region R is successfully terminated or stopped, all changes for that region must be reflected within every ancestor region P whose tree stores information about R. Completion involves an appropriate update of subplans for R within P's region plan at relevant nodes in *tree*(P), as well as an update of other relevant information about R stored at these nodes. Assuming that the final plan for R's preceding incarnation was $Rplan_i$, the nodes to be updated will be precisely those that have information about $Rplan_i$. As stated earlier, maintenance of search consistency is trickier than meets the eye. For example, completion must be coordinated with the data structures utilized by `check_affected_subregions`. If Ri and $Ri+1$ share a subregion S and search within Ri affects S, completion for S must occur in $Ri+1$'s tree before search proceeds there.

5 Complexity Analysis

It is clear that no general definitive complexity result can be given for localized search – the size and complexity of the planning search trees for a particular problem will depend on the structure of the domain, the constraints associated within each region, the complexity of their satisfaction algorithms, the domain search heuristics, and the peculiarities of the specific domain problem being solved. In order to provide some theoretical estimate of the benefits of localized search, however, we provide an idealized analysis of search for a domain with a very simple locality structure. We provide best- and worst-case search costs, assuming that constraint algorithms are either all constant, linear, quadratic, or exponential in cost (obviously, most domains will have a mixture of these). Although our analysis is quite idealized, it correlates with the empirical results of Section 6. The reader should also note that, for most of our empirical tests, search has been very close to best-case - i.e., our tests have exhibited very little backtracking. In general, best-case behavior can be expected if good domain search heuristics are employed.

To formally and empirically assess the benefits of localized search, we must compare it with completely non-localized search. For our formal complexity analysis, we utilize the non-localized and localized domain configurations depicted in Figure 4. For both domains we assume a total of n_c constraints, that each constraint has n_f possible fixes, and that the total number of actions in the final plan is s. The cost of checking any constraint on a plan of size j is $c(j)$ and the cost of fixing a plan of size j is $f(j)$. For the localized case, we assume that the domain has been localized to form a configuration of m subregions $R_1 ... R_m$ and a region G. The actions in the final plan are divided equally among the regions $R_1 ... R_m$, so that each builds a plan of size $\frac{s}{m}$. Each of the $R_1 ... R_m$ regions also contains a subregion consisting of k actions that overlaps with region G. Thus, G's region plan consists of mk actions. The n_c constraints of the original problem are evenly distributed among $G, R_1, ... R_m$ so that each region is associated with $\frac{n_c}{m+1}$ constraints.

Let us now consider the cost of a generic region search tree. Let us assume that, for a region i, there are n_{c_i} constraints, that each constraint has n_f fixes, and that the final size of the region plan is s_i. Because a constraint fix may always, in principle, violate previous constraints that may have been satisfied, constraints may need to be repeatedly



n_c constraints
n_f fixes
s plan size
$\frac{s}{n_c}$ repeat factor
Non-Localized
G
$\frac{n_c}{m+1}$ constraints
mk plan size
n_f fixes
$\frac{mk(m+1)}{n_c}$ repeat factor
k
....
k
$\frac{n_c}{m+1}$ constraints
n_f fixes
$\frac{s}{m}$ plan size
$\frac{s(m+1)}{mn_c}$ repeat factor
R1
Rm
Localized

Figure 4: Non-localized and Localized Domains

checked and fixed. The search thus tends to take the form of a round-robin checking of constraints. We call the number of times the search must cycle through the constraints the search "*repeat factor.*" Assuming that our sample region has a repeat factor of r_i, its tree depth is $r_i n_{c_i}$, with average depth to adding an action being $\frac{r_i n_{c_i}}{s_i}$. (Thus, we assume that at most one action is added per fix. In most realistic domains, many actions are often added per fix.)

To calculate search cost, we assume an implicit search space that alternately branches due to choice of a constraint (the costs $c(j)$ accumulated due to constraint testing) and choice of a fix (the costs $f(j)$ accumulated due to constraint fixing). By "best-case search" we mean depth-first search without backtracking – i.e., the cost of one path from the root to the leaves of the search space. The cost of best-case search for region i is

$$\sum_{0 \leq j \leq s_i} \frac{r_i n_{c_i}}{s_i}(c(j) + f(j)).$$

In contrast, worst-case search cost measures the cost of searching the entire space. For our sample region i this cost is

$$\sum_{1 \leq j \leq r_i n_{c_i}} n_{c_i}^j n_f^{j-1} c((j-1)\ div \frac{r_i n_{c_i}}{s_i}) + n_{c_i}^j n_f^j f((j-1)\ div\ \frac{r_i n_{c_i}}{s_i}).$$

We shall now compare the complexity of these formulae for the non-localized and localized cases. For each case, we must assume a repeat factor for each region. In general, this will be a function of the size s_i of the region plan and the number of constraints for that region n_{c_i}. For this analysis, we will set the repeat factor r_i to be $\frac{s_i}{n_{c_i}}$ – that is, we assume that exactly one action is added per fix, that the size of the plan is larger than the number of constraints, and that the depth of the region tree is equal to the number of actions in the region plan. In most of our test situations, however, the repeat factor tends to be less than this number, with more actions added per fix and, of course, some subset of actions being added by overlapping regions. Moreover, less rechecking needs to be done due to tuning of constraint application. On the other hand, some amount of additional rechecking tends to occur due to the completion process – i.e., rechecking required due to regional overlap. Thus, our assumption of a repeat factor of $\frac{s_i}{n_{c_i}}$ may be only slightly pessimistic. Given this formulation, the repeat factor for the completely non-localized case will be $\frac{s}{n_c}$. For the localized case, we have a repeat factor of $\frac{mk}{\frac{n_c}{m+1}} = \frac{mk(m+1)}{n_c}$ for region G and a repeat factor of $\frac{s}{m}/\frac{n_c}{m+1} = \frac{s(m+1)}{mn_c}$ for each of the $R_1 ... R_m$ regions.

The complexities of all cases are summarized in the tables below. We provide best- and worst-case search results, assuming that the complexity of $c(i)$ and $f(i)$ are both constant, linear, quadratic, or exponential. In some cases we supply only a leading term. For all of the localized search cases, we add to the total cost of the search trees an additional completion cost C. For this analysis, we shall assume that completion occurs each time an action is added within a region of overlap Q. The cost of each completion operation will be a function of the number of additional regions that include Q (this will not include the region actually adding the action to Q) and the size of the plan data structure for each of those regions (since completion involves update of of this data structure). In GEMPLAN, the size of the plan data structure is a function of the number of regions in the plan – in this case $m + 1$. So for this problem, we shall assume a completion cost $C = mk(m + 1)$ or $O(m^2k)$.

complexity of *c(i) and f(i)*	*Non-Localized* *(best-case)*	*Localized* *(best-case)*
constant (b)	$2bs$	$2b(s + mk) + C$
linear (ib)	bs^2	$b(\frac{s^2}{m} + (mk)^2) + C$
quadratic (i^2)	$\frac{2}{3}s^3$	$\frac{2}{3}(\frac{s^3}{m^2} + (mk)^3) + C$
exponential (b^i)	$2b^s$	$2(mb^{\frac{s}{m}} + b^{mk}) + C$

complexity of *c(i) and f(i)*	*Non-Localized* *(worst-case)*	*Localized* *(worst-case)*
constant (b)	$b(n_cn_f)^s$	$b(m(\frac{n_cn_f}{m+1})^{\frac{s}{m}} + (\frac{n_cn_f}{m+1})^{mk}) + C$
linear (ib)	$bs(n_cn_f)^s$	$b(s(\frac{n_cn_f}{m+1})^{\frac{s}{m}} + (\frac{n_cn_f}{m+1})^{mk}) + C$
quadratic (i^2)	$s^2(n_cn_f)^s$	$\frac{s^2}{m}(\frac{n_cn_f}{m+1})^{\frac{s}{m}} + (mk)^2(\frac{n_cn_f}{m+1})^{mk} + C$
exponential (b^i)	$(bn_cn_f)^s$	$m(\frac{bn_cn_f}{m+1})^{\frac{s}{m}} + (\frac{bn_cn_f}{m+1})^{mk} + C$

As can be seen in the tables, localized search is, in general, always better than non-localized search – in most cases significantly better. The only real exceptions are in the case of constant-complexity best-case search or when the cost of completion overshadows the cost of the search itself. The amount by which localized search wins over non-localized search is proportional to the amount by which s dominates both $\frac{s}{m}$ (the size of each of the subregions $R_1 ... R_m$) and mk (the size of G). Thus, increased decomposition is always worthwhile, except for the cost of increased amounts of overlap (which is reflected in the size of mk and the cost of completion C).

The overall gains of localized search increase as the complexity of the constraint algorithms increases and the bushiness (size) of the search space increases. These gains can be understood to come from three sources:

1. The absolute *size* of the localized search space is generally smaller that the non-localized search space. This becomes increasingly true as the bushiness of the search space (the amount of backtracking) increases.

2. The *cost* of localized search is cheaper than non-localized search, even if the absolute size of the search space is the same. This is because expensive constraint algorithms are applied to much smaller local plans.

3. The *search heuristics* provided by localization provide excellent guidance by helping the planner apply the most relevant constraints at a given time. Thus, the overall branchiness of localized search may be decreased simply by virtue of the good search heuristics provided by localization.

6 Empirical Results

All of our empirical experiences with GEMPLAN certainly bear out the efficacy of localized search. Our largest application so far is for a building-construction domain. This domain includes multiple instances of each type of contractor as well as multiple walls and footings to which these contractors must be allocated. The problem thus manifests both resource allocation and temporal coordination of access to building components. The application was used to test a variety of localization configurations, including some that were fairly complex, involving both a great deal of hierarchy and overlap.

The empirical tests with the construction domain have shown universal performance

Even greater speedups can thus be expected for more complex domains with bushier, more complex search spaces.

The tables below provides timing results for the construction domain (on a SPARC workstation). The "number of regions" column gives the total number of regions that have at least one constraint and one action in the final plan. The "overlap size" column gives a sum of size measures for each region of overlap. For each such region, its "size" is the number of actions in the region multiplied by the number of times it occurs within another region. For instance, in the domain of Figure 1, e-control, p-control, and wallA each occur twice within a parent region. If each region has a total of 2 actions within its plan, the domain's total overlap size would be 12. The overlap size column gives a good idea of how expensive the completion process is and the amount of rechecking that is required due to regional overlap . The "largest region" column gives a pair of numbers <*number of constraints,number of actions*> for the region with the largest number of constraints (which, in this case, is usually also the region with the largest number of actions). This measure gives an idea of how big the largest search space in the domain is – i.e., the region space in which the most search will be conducted.

The two tables below provide results for the creation of a 49-action construction plan and a 97-action construction plan. Both used the same basic domain decomposition, with the 97-action plan simply having more walls, contractors, etc. Within each table, the first test case is for a non-localized version of the domain – all constraints are applied globally. The localized(1) test configuration is highly decomposed but also has significant amounts of overlap between regions. The localized(2) case has less localization and much less overlap. Case localized(3) has an intermediate level of both localization and overlap, and attains the best results in both cases. Interestingly, these results jibe with our formal analytical results; increased localization provides increased benefit, except for the added expense caused by with regional overlap. However, notice that, in the 97-action case, localized(1) is faster than localized(2). This shows how, as plan size increases, the cost of dealing with overlap is overshadowed by the shear size of the planning space itself.

test case (49 actions)	*number of regions*	*overlap size*	*largest region*	*CPU Seconds*
non-localized	1	0	<40,49>	113.81
localized(1)	24	134	<4,16>	85.78
localized(2)	16	32	<15,28>	79.23
localized(3)	19	76	<8,17>	62.95

test case (97 actions)	*number of regions*	*overlap size*	*largest region*	*CPU Seconds*
non-localized	1	0	<52,97>	905.98
localized(1)	37	236	<7,34>	524.97
localized(2)	28	32	<24,58>	725.43
localized(3)	31	102	<8,24>	442.43

7 Conclusion

This paper has described a general-purpose technique for localized search, as well as complexity results and empirical test results that illustrate how localized reasoning can provide substantial gains in performance. I strongly believe that the principal of domain localization can be used by a wide variety of reasoning mechanisms. The idea is quite intuitive and natural, but has, surprisingly, not been a fundamental aspect of most AI systems. Its application to planning is vital if such systems are to meet the requirements of large, complex domains.

Acknowledgments

Lode Missiaen helped to design and implement GEMPLAN's localized search algorithm. I would also like to thank Anna Karlin for her assistance with the complexity results. John Bresina, Megan Eskey, Mark Drummond, Monte Zweben, Lode Missiaen, and Guy Boy also provided astute advice towards improving the quality of this paper.

References

[1] Bresina, J., Marsella, S. and C. Schmidt. "Predicting Subproblem Interactions," Technical Report LCSR-TR-92, LCSR, Rutgers University (February 1987).

[2] Chapman, D. "Planning for Conjunctive Goals," Masters Thesis, Technical Report MIT-AI-TR-802, MIT Laboratory for Artificial Intelligence, Cambridge, Massachusetts (1985).

[3] Georgeff, M.P. and A.L. Lansky. "Reactive Reasoning and Planning," in *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, Seattle, Washington (1987).

[4] Hayes, P.J. 1973. The Frame Problem and Related Problems in Artificial Intelligence. *In* Artificial Intelligence and Human Thinking. *Edited by* A. Elithorn and D. Jones. Jossey-Bass, Inc. and Elsevier Scientific Publishing Company, pp. 45-59.

[5] Knoblock, C.A. "Learning Abstraction Hierarchies for Problem Solving," in *Proceedings of the Seventh International Workshop on Machine Learning*, pp. 923-928 (1990).

[6] Korf, R.E. "Planning as Search: A Quantitative Approach," *Artificial Intelligence*, Volume 33, Number 1, pp. 65-88 (1987).

[7] Lansky, A.L. "Localized Representation and Planning," in *Proceedings of the 1989 Stanford Spring Symposium, Workshop on Planning and Search* (March 1989).

[8] Lansky, A.L. "Localized Event-Based Reasoning for Multiagent Domains," *Computational Intelligence Journal, Special Issue on Planning*, Volume 4, Number 4 (1988).

[9] Lansky, A.L. "A Representation of Parallel Activity Based on Events, Structure, and Causality," in *Reasoning About Actions and Plans, Proceedings of the 1986 Workshop at Timberline, Oregon*, M. Georgeff and A. Lansky (editors), Morgan Kaufmann Publishers, Los Altos, California, pp. 123-160 (1987).

[10] Lansky, A.L. and D.S. Fogelsong, 1987. "Localized Representation and Planning Methods for Parallel Domains," in *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, Seattle, Washington, pp. 240-245 (1987).

[11] Missiaen, L. "Localized Search," Technical Note 476, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave. Menlo Park, California 94025 (November 1989).

[12] Tate, A. "Project Planning Using a Hierarchical Nonlinear Planner," Department of

Artificial Intelligence Report 25, University of Edinburgh (1976).

[13] Wilkins, D.E. 1984. Domain-independent Planning: Representation and Plan Generation. Artificial Intelligence, Volume 22, Number 3, pp. 269-301.